AFIT/GCS/ENG/92D-16

MANAGEMENT OF SIMNET AND DIS ENTITIES IN SYNTHETIC ENVIRONMENTS

THESIS

Steven Michael Sheasby
Captain, USAF

AFIT/GCS/ENG/92D-16

AFIT/GCS/ENG/92D-16

MANAGEMENT OF SIMNET AND DIS ENTITIES IN SYNTHETIC ENVIRONMENTS

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Steven Michael Sheasby, B.S.E.

Captain, USAF

December, 1992

Acknowledgments

This thesis is part of a larger effort of many great students and faculty here at AFIT. I would like to thank my fellow students Capt Rex Haddix for his work on the synthetic battle bridge, Capt Bruce Hobbs for his work on the VISTA, and Capt Dave Tisdale for his general help. More thanks go to Capt Dean McCarty and Capt Chip Switzer for their help during the turbulent times on the virtual cockpit.

Special thanks go to my committee members, Lt Col Amburn, Lt Col Stytz, and Dr. Hartrum. These times have been some of the most trying in my life and sometimes I did not think I was going make it through graduation. Thanks very much for your help. More thanks go out to Mr Bruce Clay and Mr John Locke for the patience with me as I asked many network questions. Thanks also go out to Maj Dave Neyland and DARPA. Without DARPA their support, our synthetic environments would still be in initial development.

Steven Michael Sheasby

Table of Contents

List of Figures

List of Tables

Abstract

This thesis describes the techniques used to create an object manager utilized by an application program during a distributed interactive simulation. This work is currently utilized by a number of AFIT synthetic environment applications for use during a SIMNET exercise.

An extensive review of distributed interactive simulations is presented. A discussion of the current distributed simulation protocol, SIMNET, is presented along with the future protocol standard, DIS. Finally, a brief discussion on dead reckoning and its importance during an exercise is presented.

An analysis of the SIMNET and DIS protocols provided the basis for the creation of a series of C++ classes to store information on a simulation entity during an exercise. These C++ classes used class generalization and inheritance to differentiate between the different types of entities seen during an exercise.

An entity object manager was developed to perform a set of basic functions required during an exercise as listed in a collection of SIMNET and DIS documents. The entity object manager uses the C++ entity class structure to manage the numerous entities viewed during a typical SIMNET exercise. The entity object manager also communicates with the the other exercise participants using two different government supplied network communications packages.

MANAGEMENT OF SIMNET AND DIS ENTITIES IN SYNTHETIC ENVIRONMENTS

I. Introduction

1.1 Overview

Modern warfare is the one of the most complex activities performed by humans. An important aspect of modern warfare is ability to interact and coordinate with other personnel during an engagement. Thus training for teamwork and coordination is crucial (46:492).

Until the present time, the United States Armed Forces have relied on field exercises to bring together the teamwork and coordination aspects of warfighting (46:492-493). However, field exercises have limitations. These exercises are extremely expensive, especially in an era of dwindling defense budgets. Also, exercises must be weighed against any environmental concerns (10:1).

The United States Armed Forces has already developed stand-alone simulators and training systems for major weapon system purchases (25:119). In the 1980s, the Defense Advanced Research Project Agency (DARPA) developed a system to interconnect these simulators for use in tactical team training. The system was called the Distributed Simulator Networking (SIMNET) program (29:1). SIMNET allowed information produced by a simulator, such as its location and its status, to be broadcast over a local or long-haul network and picked up by other simulators participating in that exercise (26:136).

Since 1988, thesis students at the Air Force Institute of Technology (AFIT) have been researching different aspects of low cost flight simulators and other related virtual reality systems (40:1-1). In 1991, DARPA requested that AFIT investigate the possibility of a low cost virtual cockpit to be used during large scale interservice war gaming SIMNET simulations. The cockpit

would be composed of inexpensive graphics workstations (less than $250,000), off-the-shelf software and hardware components (47:147), and AFIT developed software.

1.2 Problem Statement

The focus of this thesis effort was the software design and implementation of a real-time SIMNET object manager for the virtual cockpit and the synthetic battlebridge. The SIMNET object manager broadcasts information about the virtual cockpit while receiving information from other entities involved in a simulation. The SIMNET object manager converts the information from the SIMNET protocol into a form usable by the virtual cockpit and the synthetic battlebridge.

The SIMNET object manager consists of three sections:

- An interface to government supplied network broadcasting software.
- A series of C++ classes for the various simulation objects, such as aircraft, tanks, and missiles. These classes manage information that is received over the network during a simulation.
- An entity object manager that inserts, updates, and deletes simulation entities based on the above classes.

1.3 Approach

1.3.1 Assumptions An object oriented analysis and design of the software was required for creation of the entity class structure and the entity object manager. The object model has advantages over traditional methods such as encouraging software reuse, appealing to way the human mind naturally works, and producing systems built upon stable intermediate forms (14:1-5).

Other system requirements included:

- Target Machine - Silicon Graphics IRIS 4D Series (multiprocessor)
- Operation System - UNIX (IRIX 4.0.x on the Silicon Graphics)
- Programming Languages - AT&T C++, and C (when necessary)

The system was to be developed in a series of prototypes, with each prototype adding functionality to its predecessor.

1.3.2 Entity Classes The entity class structure was to be developed using the current SIMNET protocol (31) and the future Distributed Interactive Standard (DIS) protocol (27) using the C++ language. The information contained in the protocol packets determined the breakdown of information in the entity class structure. These classes are discussed in detail in Chapter III.

Generalization was the concept used in creating the class structure. Each level down in the hierarchy inherited the attributes and methods from the level above while adding its own specific attributes and methods (34:38-43).

1.3.3 Entity Object Manager The entity object manager was to be developed using only the current SIMNET protocol (31). The entity object manager inserts, updates, traverses, and deletes entities having information corresponding to the appropriate entity class structure. The entity object manager also dead reckoned any moving vehicles.

The entity object manager also started the network communications daemons, terminated any network communications daemons, received SIMNET packets from other simulators, and broadcast SIMNET packets about our virtual cockpit. The entity object manager also contained methods for communicating with other modules from other thesis software, such as the synthetic battlebridge.

The entity object manager was developed to use either of two separate SIMNET network communications packages. The first package was developed by John Locke at the Naval Postgraduate School (NPS) in Monterey, CA. This package broadcasts and receives raw Ethernet messages.

All programs managing raw Ethernet packets requires root privileges. This method of broadcasting is required during exercises with other simulators.

The second package was developed locally by Mr Bruce Clay. This package broadcasts and receives UDP packets. UDP packets can be received and sent without root privileges. This communications package is used to reduce the access to root privileges during software development and testing.

The initial network requirements called only for use of the SIMNET network communications package developed at NPS. However, the additional requirement of a locally developed package not requiring root privileges was added to reduce the need for students to test their applications while running at root.

1.3.4 WAR BREAKER Requirements As stated previously, the SIMNET protocol formed the basis for creating the entity class structure and for some of the details in the SIMNET object manager. However, the virtual cockpit was developed for initial use in the WAR BREAKER program. The WAR BREAKER program modified the SIMNET protocols to add additional information not included in the original protocols (37).

In addition, WAR BREAKER assumed a simulated gaming grid of northwest Iraq. Display of both SCUD and Patriot missiles, missile launchers, and support vehicles were required. Both SCUD and Patriot launches had to be displayed.

1.4 Additional Thesis Support

1.4.1 Virtual Cockpit Along with two other thesis efforts, this thesis forms the basis for the virtual cockpit. The two other thesis efforts are used to support different sections of the virtual cockpit:

- Capt Switzer's thesis effort resulted in the virtual display of a cockpit layout. Capt Switzer's software received data from three input devices to the virtual cockpit: a throttle and stick combination for airplane control and the Polhemus tracker for head position. The software also broadcast signals to the Head-Mounted Display (HMD). Capt Switzer's software also contained the appropriate flight dynamics (45).

- Capt McCarty's thesis effort resulted in the display of terrain information from a number of terrain databases. Capt McCarty's software also displayed the objects that were in the simulated world.

1.4.2 Synthetic Battlebridge Along with Capt Haddix's thesis effort, this thesis forms the basis for synthetic battlebridge. The synthetic battlebridge is a virtual representation of a battlefield command and control room (14:1-1).

Capt Haddix's thesis software received input from the Polhemus tracker for head position and the Apple Macintosh for voice commands. The software broadcast signals to the Head-Mounted Display (HMD). Capt Haddix's software displayed the objects in the simulated world on a rectangular section of terrain.

1.4.3 VISTA Capt Hobbs thesis effort was the display of the battlefield using a Texas Instruments Omniview three-dimensional display. Capt Hobbs software displayed the objects in the simulated world on the Omniview. The software also displayed information about the simulated world using a series of windows on the screen on the Sun Sparcstation (18). The high level communications subroutines were provided to Capt Hobbs to allow interaction during a simulation.

1.5 Thesis Overview

Chapter II describes distributed interactive simulations. SIMNET, DIS, and dead reckoning are discussed in that chapter. Chapter III describes the design and C++ implementation of the entity class structure. Chapter IV describes the design and C++ implementation of the entity object manager. Chapter V contains results and conclusions.

II. Background

This chapter discusses some recent developments in distributed interactive simulations and how they apply to this thesis work. Also discussed are two of the distributed simulation message protocols: SIMNET and DIS. Finally, how dead reckoning is used to reduce network bandwidth during simulations is discussed.

2.1 Distributed Simulations

As the United States Armed Forces acquires a new weapons system, weapon system simulators are also purchased. This is especially true in the United States Air Force. There are many benefits for flight simulators (40:1-1). Safety is one of the most important concerns. It is much safer to fly a simulator than fly an actual airplane. The cost for flying a simulator is much less than the actual flying (12:919). In addition, a simulator can be made available the entire day while training flights are limited due to availability of aircraft and weather conditions. These same concerns apply to Army vehicles, such as the M1 Abrams tank or the Apache Attack Helicopter.

Simulators are useful for training personnel to perform their duty on an individual weapon systems (25:119). However, another area of training is the area of team coordination. The United States found during the military operations in Grenada and Panama that problems existed when performing coordinated combined arms force (10:1). Field exercises, multi-service exercises, and multi-national exercises have been developed as means to develop coordination among the different elements during battle (25:119).

These exercises have some problems. A major factor is the cost of such exercises, both in terms of money and supplies for the participants (25:119). Lack of adequate exercise locations mean that there exist long time periods between exercises. Other factors include vehicle maintenance, transportation of the vehicles, and exercise ammunition limits (33:347).

In order to perform component level training when exercises were not feasible, an approach that linked (or networked) the various stand-alone simulators together was necessary (1:91). This approach is the foundation of distributed simulations.

Distributed simulator systems have a number of advantages (1, 8):

- The number of simulators participating in an exercise is not limited. The only limits are the number of simulators available and possible network traffic limitations.
- The linked simulators are not homogeneous. For example, an M1 Abrams tanks simulator and an F-15 flight simulator can be linked and can interact with each other.
- The distributed simulation system is flexible. Simulators can be added, removed, or even changed without disruption of the distributed simulation system. In fact, the changes would be transparent to the other simulators.
- The simulators do not need to be physically co-located with each other. By use of local or long-haul communications networks, simulators would be able to participate in an exercise as if they were next to each other.

Distributed simulator systems also have a number of possible problems (1, 9):

- A network communication link failure could be a major problem during a simulation. How the local simulators handle network failure is important.
- Network bandwidth is a limiting factor. As the number of simulators grow on the simulation network, the amount of network traffic increases. The rate of increase is dependent on the types of vehicles being simulated and the type of actions performed during an exercise. A saturation point may be reached where adding another simulator will result in loss of some network traffic. Dead reckoning techniques help alleviate that problem.
- Network latency is another important issue. Latency is the elapsed time to move a packet from one point to another. A certain amount of latency is expected in message traffic. If the latency is high, messages could be useless by the time they are received by the other simulators.
- All simulators need to communicate their information, such as location and direction, in the same format. The creation of the SIMNET and DIS standards were developed to minimize this problem.

The basic terrain and cultural objects are assumed to be known to every simulator participating in an exercise (29:1). This implies that details of an exercise are set up in advance of the start of that exercise. In addition, each simulator is responsible for keeping track of information about the other entities in a simulation (29:2). Most of this information usually comes from an initial message about an entity (31:81-85).

In addition to manned vehicle simulators, distributed simulation networks allow for other modules to be connected during an exercise. These additional types include support and combat service vehicles, command post simulators, semi-automated forces, and data collection and analysis systems (29, 28).

During an exercise, it is important to be able to accurately render the movements of the various combat vehicles. However, during a realistic engagement, support vehicles play a vital role in the outcome of a battle. Simulators exist, such as the SIMNET Management, Command and Control (MCC) system that control the various support vehicles. These support vehicles include general cargo trucks, refueling vehicles and ammunition resupply vehicles (28:578).

Command post simulators focus on the information necessary to decision making at the highest level of battlefield management. Instead of an out-the-window display, a display or displays of the overall battlefield are used.

Semi-automated forces (SAF) simulators allow one person to manipulate a large number of vehicles in a simulation from a workstation or series of workstations (39:1). A human commander provides the goals and objectives for the subordinate units that comprise the SAF simulator. The simulator makes choices for unit movement, route planning, obstacle avoidance, and target engagement. The human commander has the option of taking direct command of a subordinate unit when appropriate (29:3).

Data collection and analysis systems capture onto a disk file every action taken during an exercise. Each event, such as movement and weapons fire, are timestamped and logged to a disk

file. This disk file can be replayed at a time in the future (28:578). The BBN SIMNET data logger (38) and the Loral Table Logger (41) programs are examples of such systems.

As stated above, an important aspect of distributed simulation is a standard protocol language for transmitting of vehicle state information. Two standards exist today: the SIMNET protocol and the proposed DIS standard.

2.1.1 SIMNET In 1983, the Distributed Simulator Networking (SIMNET) program was started by the Defense Advanced Research Projects Agency (DARPA) and the U.S. Army. One of the goals of SIMNET was to develop the technology to build a cross-country network of interactive combat simulators (31:i).

That goal has been achieved as approximately 250 simulators have been delivered to the U.S. Army, including M1 Abrams and M2 Bradley simulators delivered to Ft. Knox, KY and helicopter simulators at Ft. Rucker, AL (29:1)

A set of standard rules and conventions were created for simulators to exchange information quickly and efficiently over the communications network. For the SIMNET project, these rules are called the SIMNET protocols (31:1) The SIMNET protocol is composed of (31:10-11):

- An underlying communication service, such as Ethernet,
- An association protocol supporting distributed simulations,
- A simulation protocol used to convey information about elements in the exercise, and
- A data collection protocol used to report exercise-level information.

A SIMNET message consists of the data included by the communication service, the association protocol, and either a data collection protocol or simulation protocol.

A SIMNET simulated world is a rectangular terrain grid with axes labeled X, Y, and Z. The positive X axis points east, the positive Y axis points north, and the positive Z axis points upward. This is known as the world coordinate system. The SIMNET origin (0, 0, 0) is the southwest corner of the terrain grid. To maximize the precision at which locations can be expressed, most of the simulation takes places as close to the origin as possible. The unit of measure in SIMNET is the meter (31:15-16).

A vehicle in the SIMNET simulated world also has the same axes orientation (31:16). This is known as the vehicle coordinate system. The X axis points to the vehicle's right, the Y axis points to the vehicle's front, and the Z axis points upward. The origin depends on the type of vehicle. For a tank, the origin is the center of the vehicle's base.

A SIMNET vehicle uses a 3 × 3 rotation matrix to express its orientation with relation to the vehicle coordinate system (31:16). This matrix can be considered as the result of successive heading, pitch, and roll rotations about the vehicle coordinate system. Heading is a negative rotation about the z axis, pitch is a positive rotation about the x axis, and roll is a positive rotation about the y axis. The resultant matrix is

$$\begin{vmatrix} M_{0,0} & M_{0,1} & M_{0,2} \\ M_{1,0} & M_{1,1} & M_{1,2} \\ M_{2,0} & M_{2,1} & M_{2,2} \end{vmatrix} \quad (1)$$

where

$$M_{0,0} = cos(heading) * cos(roll) + sin(heading) * sin(pitch) * sin(roll) \quad (2)$$

$$M_{0,1} = -sin(heading) * cos(roll) + cos(heading) * sin(pitch) * sin(roll) \quad (3)$$

$$M_{0,2} = -cos(pitch) * sin(roll) \quad (4)$$

$$M_{1,0} = sin(heading) * cos(pitch) \quad (5)$$

$$M_{1,1} = cos(heading) * cos(pitch) \quad (6)$$

$$M_{1,2} = sin(pitch) \quad (7)$$

$$M_{2,0} = cos(heading) * sin(roll) - sin(heading) * sin(pitch) * cos(roll) \quad (8)$$

$$M_{2,1} = -sin(heading) * sin(roll) - cos(heading) * sin(pitch) * cos(roll) \quad (9)$$

$$M_{2,2} = cos(pitch) * cos(roll) \quad (10)$$

The basic unit for the SIMNET simulation and data collection protocols is the Protocol Data Unit (PDU). The data collection and simulation protocols have their own sets of different PDUs (31:11).

Data collection PDUs are used to report information about the simulated world in SIMNET exercises. Data collection PDUs provide information useful to analysts studying an exercise and preparing the system for restart after any unexpected interruptions (31:118).

Simulation protocol PDUs are used to enter entities into an exercise, withdraw entities from an exercise, and describe the appearance of the entity. Simulation protocol PDUs also report firing and impact of projectiles, transfer supplies to entities, and repair entities (31:76). There are 19 different types of simulation protocol PDUs. Table 1 lists the different types.

The Vehicle Appearance PDU is primary source of data exchange in SIMNET. This PDU contains information about an entity: object type, location, speed, and appearance (31:88). A majority of the PDUs broadcast during an exercise will be Vehicle Appearance PDUs. At a minimum, an entity will broadcast a Vehicle Appearance PDU every 5 seconds (31:88). Dead reckoning will decrease the number of times Vehicle Appearance PDUs are broadcast.

Activate Request	Activate Response
Deactivate Request	Deactivate Response
Vehicle Appearance	Radiate
Fire	Impact
Indirect Fire	Collision
Service Request	Resupply Offer
Resupply Received	Resupply Cancel
Repair Requested	Repair Response
Marker	Breached Lane
Minefield	

Table 1. SIMNET Simulation Protocol PDUs

There are two ways to enter an entity into a SIMNET simulation. One is using the Activate Request PDU. This PDU allows an active entity to activate a dormant entity - such as one vehicle towing another vehicle (31:81-85). The Activate Response is used by the new entity to reply to the request (31:86). The other method is for a new entity to issue a Vehicle Appearance PDU.

There are also two ways to remove an entity from a SIMNET simulation. One is using the Deactivate Request PDU. This allows an entity to broadcast to the rest of the players that the entity is leaving the simulation (31:86-87). The Deactivate Response PDU is broadcast by another simulator if it controls the original entity (31:87-88). The other method is simply to stop broadcasting Vehicle Appearance PDUs. If other entities do not receive Vehicle Appearance PDUs within 12 seconds of each other on a single entity, then that entity is removed from the simulation (31:88).

Other PDUs describe events that occur during an exercise. A Radiate PDU is issued by a simulator that uses its own radar during an exercise (31:93). A Fire PDU is issued by a simulator firing a shell, burst of machine gun, or missile (31:101). An Impact PDU is issued when the flight of the projectile it is simulating terminates (31:103).

The underlying communication service used in SIMNET are either of two Ethernet standards (31:171-174). The preferred standard is the IEEE 802.3 Ethernet standard (4, 13, 2, 3). The older Ethernet standard, Ethernet Version 2.0, can be used.

There are some problems with SIMNET in the current distributed simulation environment. The SIMNET Vehicle Appearance PDU field that describes object types was developed for Army vehicles only. Some aircraft have been inserted into the object types as later versions of the protocol have been created, but the number is limited (only F-16, A-10, and F-14 for United State Air Force and Navy aircraft). The Naval Ocean Systems Center had problems adapting the SIMNET protocol for their Battle Force Inport Trainer (5, 6). The current object type field can not properly accommodate Navy vehicles (6:3-5).

Another major drawback is the fidelity of representing high performance vehicles, such as fighter aircraft. In order to model an aircraft's new position and orientation with the highest possible fidelity, linear acceleration and angular velocity are required for the harmonic dead reckoning model (see the dead reckoning section at the end of this chapter for a discussion of the various dead reckoning models). The SIMNET protocols do not contain this information in any of the PDUs. Only the simplest of dead reckoning models are allowed.

In addition, new PDUs are needed to represent some of today's exercise elements such as voice and data communication and atmospheric conditions. This produced an effort to create a new standard for use in distributed simulations: the Distributed Interactive Simulation (DIS) standard.

2.1.2 Distributed Interactive Simulation (DIS) The DIS standard can be considered as the follow-on to the SIMNET protocol descriptions. In 1989, DARPA and Army Project Manager for Training Devices (PM TRADE) started the process to develop a new distributed simulation standard (25:120). The result was the Distributed Interactive Simulation (DIS) protocol.

Due to the success of the SIMNET approach, DIS incorporates all the essential elements of SIMNET into the new standard (29:1). For instance, the essential information of the Vehicle Appearance PDU in SIMNET is included in the Entity State PDU in DIS.

DARPA and PM TRADE decided that the best way to create this new standard was to utilize

a series of workshops which allowed developers and users to create solutions to common problems (11:1). A set of working groups was established in the different areas of distributed simulations. Each working group established subgroups to handle specific tasks (11:2). The working groups announce their results at the semi-annual DIS workshops. The results of the workshops are the set of documents that produced the DIS standard.

One important aspect of this DIS standard development effort is the effort to recognize DIS as an international standard for distributed simulation. Thus, the DIS standard has been submitted to the Institute of Electrical and Electronic Engineers (IEEE) for standard approval (10:9). Following IEEE approval, the standard will be submitted to the appropriate international agencies for international standard approval. This is useful since United States allies can utilize the standard for their simulators (10:9).

Since DIS is considered a direct descendent of SIMNET, DIS inherited most of its implementation principles from SIMNET. DIS also added its own principles.

These are the ideas DIS inherited from SIMNET (43, 32):

- SIMNET and DIS do not need a central computer to control the simulation by scheduling events or conflict resolution.
- SIMNET and DIS consist of a distributed simulation environment using local copies of common terrain and models database.
- Each SIMNET and DIS entity maintains its own world view based on its own simulation.
- Each SIMNET and DIS entity is responsible for determining what is perceived.
- Each SIMNET and DIS entity only broadcast packets that demonstrate changes in their state.
- Each SIMNET and DIS entity employs dead reckoning to reduce communications processing.
- Each SIMNET and DIS entity closely corresponds to the weapon systems that they are simulating.

Additional DIS implementation principles were added (43:10):

- The DIS architecture expands the standard to include a wider range of heterogeneous simulators than SIMNET.
- The DIS architecture must support software reuse.
- The DIS architecture must support openness as its primary objective.

In SIMNET, the basic unit in an exercise is a vehicle. In DIS, the basic unit is called an entity. Thus, entity and vehicle are synonymous in this discussion.

DIS uses a different coordinate systems than SIMNET. The world coordinate system is based on the World Geodetic Survey 1984 (WGS84) (32:22). The origin is the centroid of the earth, with the positive x-axis passing through through the Prime Meridian at the equator, the positive y-axis passing through 90 degrees East longitude at the equator, and the positive z-axis passing through the North Pole. The basic unit of measure is meters (27:43).

Each entity has its own entity coordinate system. The positive x-axis extends out the front of the entity, the positive y-axis extends out the right side of the entity, and the positive z-axis points downward. The origin is the center of the bounding volume of the entity (27:42).

Unlike SIMNET, DIS uses Euler angles (ψ, θ, and ϕ) to represent the entity's orientation with respect to the entity's coordinate system (27:38). Euler angles can be described as a series of successive rotations about the three axes (22:6.1):

- first, yawing by angle ψ until the final heading is reached,
- second, pitching by angle θ until the final pitch altitude is reached, and
- finally, rolling by angle ϕ until the final bank is reached.

Entity State	Fire
Detonation	Service Request
Resupply Offer	Resupply Received
Resupply Cancel	Repair Complete
Repair Response	Collision

Table 2. DIS Protocol Data Units

Activate Request	Activate Response
Deactivate Request	Deactivate Response
Emitter	Radar

Table 3. DIS Interim Protocol Data Units

The values of ψ and ϕ range from $-\pi$ to π. The value for θ has a range from $-\frac{\pi}{2}$ to $\frac{\pi}{2}$.

Like SIMNET, the basic unit of communication in DIS is the Protocol Data Unit (PDU). Version 1.0 of the standard contains a series of required PDUs and a series of interim PDUs (27). Required PDUs are listed in table 2. Interim PDUs are listed in table 3. The required PDUs are used to support combat interactions. The interim PDUs are suggested as addressing simulation control and additional battlefield environments (32:42).

The Entity State PDU is the equivalent Vehicle Appearance PDU in SIMNET. The Entity State contains information about an entity's state during an exercise. The Entity State contains information on an entity's location, linear velocity, orientation, appearance, and capabilities (27:47-50). Like the Vehicle Appearance PDU in SIMNET, the Entity State PDU will constitute the majority of PDUs broadcast during an exercise.

The future network communications service for DIS will be the Government Open Systems Interconnection Protocol (GOSIP) (10:8). GOSIP is the United States implementation of the Open Systems Interconnection (OSI) Reference Model developed by the International Organization for Standards (42:389-399). Since GOSIP is still under development, DIS will use standard, commercially available communications protocols, such as UDP/IP (42:518-519), in the interim (10:8).

DIS, like SIMNET, has a broadcast mode. A system in broadcast mode will transmit to all systems receiving messages without knowing who are the receivers. If a packet needs to be sent to all participants in a simulation, the packet is sent using broadcast mode. DIS also adds multicast mode. Multicast mode is similar to broadcast mode except that the receivers are known to the sender (30:101). If a packet is sent to only certain participants, such in the case of dual exercises taking place at once, the packets are sent using multicast mode.

2.1.3 Dead Reckoning Dead reckoning is one of the important concepts in distributed simulation. The motivation for use of dead reckoning is reduction of the network bandwidth required to support a given application (17:128). The simulator could issue a Vehicle Appearance PDU or Entity State PDU whenever the vehicle's status changed. However, the status would change with every recomputation of the vehicle's location or velocity (31:18). The status would change at the frame rate of the simulation. With hundreds of vehicles participating, the network traffic would become intolerable.

Dead reckoning allows the simulator to reduce the frequency of Vehicle Appearance or Entity State PDUs. Dead reckoning works for both incoming and outgoing messages. The simulator must keep internal records of all the other vehicles participating in the simulation. If no Vehicle Appearance or Entity State PDU is received when the simulator updates the world, the simulator would take the last known position and velocity of the entity, and linearly extrapolate the new position. The simulator would then insert this new position into the entities' record (31:19).

The simulator keeps an internal record of its own vehicle status. The simulator also maintains a dead reckoned internal record of its location and orientation. When the dead reckoned model exceeds some threshold - the difference between the dead reckoned model and the actual model - a new Vehicle Appearance or Entity State PDU is broadcast. The dead reckoned record is updated with the new information, and the process starts over again (31:19).

Dead reckoning is a tradeoff between three factors: network bandwidth requirements, computational power needed to dead reckon entities, and precision of the entities location and orientation (31, 17). The additional computational power to dead reckon entities is small compared to reduction of network traffic using dead reckoning. The fidelity of the entities is a function of the threshold and the choice of dead reckoning algorithm (17:128). As thresholds decrease, the network traffic increases. Using higher order dead reckoning algorithms increases the network traffic. What is optimum is dependent on what vehicle is simulated.

There are three different dead reckoning algorithm. They are combinations of dead reckoning algorithms for location and dead reckoning algorithms for orientation (17:131). Different algorithms produce various degrees of accuracy of dead reckoned position and orientation. The proper algorithm is chosen depending on the vehicle to be dead reckoned.

For position, the dead reckoning algorithms are zero, first, and second order reckoning of position (35, 7).

- Zero order dead reckoning of location simply means location was maintained constant over the frame interval.

- First order dead reckoning of position consists of simply integrating linear velocity over the frame interval. The equations for first order dead reckoning position are

$$x' = x + V_x t \tag{11}$$

$$y' = y + V_y t \tag{12}$$

$$z' = z + V_z t \tag{13}$$

  where x, y, and z are the current position in three dimensional coordinates, x', y', and z' are the new position coordinates, t is time in seconds, and V_x, V_y, and V_z are the components of the velocity vector.

- Second order dead reckoning of position consists of integrating acceleration over the frame interval to update velocity and then integrating velocity to yield position. The equations for second order dead reckoning position are

$$x' = x + V_x t + \frac{1}{2} A_x t^2 \tag{14}$$

$$y' = y + V_y t + \frac{1}{2} A_y t^2 \tag{15}$$

$$z' = z + V_z t + \frac{1}{2} A_z t^2 \tag{16}$$

where A_x, A_y, and A_z are the components of the acceleration vector.

For orientation, the dead reckoning algorithms include either zero order or first order reckoning of orientation(35, 7).

- Zero order dead reckoning of orientation simply means orientation was maintained constant over the frame interval.

- First order dead reckoning of orientation consists of integrating angular velocity over the frame interval to update orientation. The formula is a combination of additions and multiplications resulting in a 3 × 3 matrix. The Orientation vectors are 3 × 1 matrices containing the values for ψ, θ, and ϕ (25:176-178). The simplification of the formula is:

$$Orientation' = (I cos\theta + \vec{a} sin\theta + \vec{a}\vec{a}^T(1 - cos\theta)) * Orientation \tag{17}$$

where I is the identity matrix, $\vec{a}$ is a matrix composed of angular velocity vectors, and θ is a Euler Angle Record component representing current orientation (as are ψ and ϕ).

There are a number different dead reckoning models currently used in distributed simulations. These models result from different combinations of position and orientation dead reckoning algorithms (35:A-134). These dead reckoning models are:

- *Static Model.* This model is used for objects that do not move under normal circumstances.

- *Linear Model.* This represents vehicles for which position is independent of orientation. Only position and linear velocity are used to determine new position. Orientation remains constant.

- *Coordinated Model.* This also represents vehicles for which position is independent of orientation. Position, linear velocity, and linear acceleration are used to determine new position. Orientation remains constant.

- *Harmonic Model.* This represents vehicles for which position and orientation are not independent. Position, linear velocity, and linear acceleration are used to determine new position. Orientation used angular velocity to determine new orientation.

Dead Reckoning Model	*Position Algorithm*	*Orientation Algorithm*
Static	Zero	Zero
Linear	First	Zero
Coordinated	Second	Zero
Harmonic	Second	First

Table 4. Dead Reckoning Models

Table 4 summarizes these dead reckoning models.

The current SIMNET standard only supports the linear dead reckoning model. Orientation was not important since it can be determined by its location on the terrain. For the SIMNET standard modified for the War Breaker exercises, all dead reckoning models are available (37:8-10). DIS supports all dead reckoning models (27:175-178).

Many studies have been performed trying to determine which algorithm is best under a given set of circumstances (15, 21). A number of reports have focused on fixed-wing aircraft and dead reckoning (16, 17, 36). The harmonic model produces the highest fidelity among the dead reckoning models. However, the harmonic model requires the most calculations making the model computationally intensive compared to the other dead reckoning models. The use of model is best determined by the workstation, type of vehicle being simulated, and the possible network traffic. The key is that there was a decrease in network traffic no matter which dead reckoning model used.

2.1.4 Summary This chapter discussed the background on distributed interactive simulations. There are many similarities between the SIMNET and DIS protocols, such as the type of information broadcast during an exercise. However, there are also many differences between the two protocols, such as the variety of vehicles able to represented during an exercise. Dead reckoning is an important part of both SIMNET and DIS.

III. Simulation Entity Classes

3.1 Introduction

This section explains the process used to create the entity class structure used by the AFIT virtual cockpit and the synthetic battlebridge. Type structures from SIMNET are analyzed to determine the various fields of information available about an entity. This analysis was used as the basis for the entity class structure.

The object oriented design technique presented by James Rumbaugh, et al in *Object-Oriented Modeling and Design* was chosen to create the entity class structure. This technique was chosen by the student due to familiarity from previous courses.

3.2 Protocol Analysis

3.2.1 SIMNET Analysis Every SIMNET simulation PDU contains fields that describe the type of vehicle the workstation is simulating. SIMNET differentiates between different objects with the **guise** field in the Activate Request PDU and the Vehicle Appearance PDU. The **guise** field is of the SIMNET type **VehicleGuises**. **VehicleGuises** is a record (31:43):

```
type VehicleGuises sequence {
   distinguished    ObjectType,
   other            ObjectType
}
```

The **VehicleGuises** has two fields (**distinguished** and **other**) to allow a vehicle to be viewed differently among difference forces (31:43). However, the **other** field of the record contains the same value as the **distinguished** .eld since no SIMNET simulators currently utilize two different vehicle appearances. Only the **distinguished** field is useful during exercises.

ObjectType is a 32-bit record that describes attributes about a vehicle. These attributes include country, vehicle series (such as F-16), vehicle model (such as F-16A), and function (31:36).

Other
Vehicle
Munition
Structure
Life Form

Table 5. `ObjectType` Domain Field Values

Domain: Vehicle	*Domain: Life Form*
Domain	Domain
Environment	Type
Class	
Country	
Series	
Model	
Function	

Table 6. `ObjectType` Record Formats

The `ObjectType` record is interpreted as a series of smaller fields, spanning a consecutive number of bits. The `ObjectType` record is broken down from left (most-significant bit) to right (least-significant bit) (31:148). The first three consecutive bits comprise the domain field. The possible values for the domain field are shown in Table 5.

For each type of domain, the other 29 bits are interpreted differently. Table 6 shows the format of `ObjectType` record for a domain value of vehicle and a domain value of life form. There is no `ObjectType` record definition for structures.

The munition domain is more complex. The next four bits in the `ObjectType` record for the missile domain describe the munition class, such as missile, projectile, and mines (31:151-152). The munition class determines the structure for the remainder of the `ObjectType` record. The differences among the different munition classes are the functionality between the fields. Table 7 shows the various `ObjectType` record definitions for the munition classes defining a `ObjectType` structure: ammunition, missile, bomb, and mine (31:152-157). The munitions classes detonator, projectile, and propellent are combined into ammunition. The munition class petroleum, oil and

Domain: Munition	*Domain: Munition*	*Domain: Munition*	*Domain: Munition*
Munition Class: Ammunition	*Munition Class: Missile*	*Munition Class: Bomb*	*Munition Class: Mine*
Domain	Domain	Domain	Domain
Class	Class	Class	Class
Caliber	Target	Weight	Target
Subclass	Warhead	Subclass	Environment
Country	Country	Country	Country
Series	Series	Series	Series
Model	Model	Model	Model

Table 7. `ObjectType` Record Format for Munition Domain

Miscellaneous
Air
Ground
Space
Water

Table 8. Vehicle Environment Field Values

lubricants has no `ObjectType` definition.

For the vehicle domain, the `ObjectType` record contains an environment field. This field describes the environment in which the vehicle operates (31:149). The values for this field are listed in Table 8. No space or water vehicles are defined in the SIMNET document, just air and ground vehicles.

Another source of information, specifically about a vehicle, is part of the Vehicle Appearance PDU record. The Vehicle Appearance PDU record field `vehicleClass`, which is of type `VehicleClass`, determines whether a vehicle has any independently moving parts. The possible values are listed in Table 9. These classes also have a bearing on the dead reckoning algorithm. The three classes are defined (31:20):

- `static` Vehicles are always stationary, have no need for dead reckoning, and have no independently moving parts.

Irrelevant
Static
Simple
Tank

Table 9. Vehicle Class Values

- **simple** Vehicles may move, use linear dead reckoning model, and have no independently moving parts.

- **tank** Vehicles may move, use linear dead reckoning model, and have an independently moving turret and gun barrel. The M1 tank and M2 fighting vehicle are both tank class members.

For a simple vehicle class, the last field of a Vehicle Appearance PDU is (31:89):

```
    ...                    ...
    velocity               VelocityVector,
    }
  }
}
```

However, for a tank class, the last fields of a Vehicle Appearance PDU are (31:89):

```
    ...                    ...
    velocity               VelocityVector,
    turretAzimuth          Angle,
    gunElevation           Angle,
    }
  }
}
```

The WAR BREAKER SIMNET protocols added the following fields to a Vehicle Appearance PDU for the simple vehicle class (37:8):

```
      ...                    ...
  linearAcceleration     LinearAccelerationVector,
  angularVelocity        AngularVelocityVector,
  throttlePosition       Float,
  fuelQuantity           Float,
}
```

The rest of the Vehicle Appearance PDU record contains data that is necessary to maintain state information on other entities involved in an exercise:

- **vehicleID** field of type **VehicleID** - Each vehicle participating in an exercise has a unique vehicle identifier. The **vehicleID** is the SIMNET identifier. The VehicleID record is composed of three elements: site, host, and vehicle. Each SIMNET site has a unique identifier. Each individual machine at a site has a unique identifier. Each vehicle generated by a particular machine has a unique identifier. This produces a unique **vehicleID** for each SIMNET entity (31:43).

- **force** field of type **ForceID** - This field identifies which force, such as observer and target, this vehicle is assigned (31:34-35).

- **location** field of type **WorldCoordinates** - This field describes the location of a vehicle in X, Y, and Z coordinates in the SIMNET world coordinate system (31:53).

- **rotation** field of type float. This is 3 x 3 rotation matrix for each entity (31:89-90).

- **appearance** bit field. This field describes some basic appearance characteristics for an entity (31:90-91). The individual elements are described in Table 10.

- **marking** field of type **VehicleMarking**. This contains the text that would appear on the entity (31:44).

- **timestamp** field. This field contains the time that the simulator broadcast the Vehicle Appearance PDU relative to that workstations' simulation time (31:90).

- **capabilities** field of type **VehicleCapabilities**. This field contains information on whether the vehicle can supply certain services to other simulated vehicles (31:41). These function are listed in Table 11.

- **engineSpeed** of type integer. This field contains the vehicle's engine speed in revolutions per second. This is useful to simulators that can synthesize sounds produced by nearby entities (31:92).

3.2.2 DIS Analysis The DIS Entity State PDU contains all the information necessary to keep track of another entity participating in an exercise. The Entity State PDU contains the same information as a SIMNET Vehicle Appearance PDU with a few exceptions.

Field	*Comment*
Destroyed	
Smoke Plume	
Flaming	
Dust Cloud	
Mobility Disabled	
Fire Power Disabled	
Communications Disabled	
Shaded	
TOW Launcher Up	Specific to M2 and M3 Bradley
Engine Smoke	Specific to T72 tank
Position Mask	Specific to life form

Table 10. SIMNET Vehicle Appearance PDU **appearance** Field Items

Ammunition Supply
Fuel Supply
Recovery
Repair

Table 11. SIMNET Vehicle Appearance PDU **VehicleCapabilities** Record Elements

The DIS Entity State PDU contains the **Entity Appearance** field. This field is similar to the SIMNET **appearance** field. A difference is the format for the **Entity Appearance** field changes with the different types of vehicles. The values for those different types of vehicles and the **Entity Appearance** field structure is listed in Table 12 (27:129-134).

Land Domain	*Air Domain*	*Water Surface Domain*	*Water Sub-Surface Domain*
Destroyed	Destroyed	Destroyed	Destroyed
Smoke Plume	Flaming	Flaming	Wake
Flaming	After Burner	Wake	Running Lights
Dust Cloud	Running Lights	Running Lights	Hatch
Paint Scheme	Navigation Lights		
Launcher	Formation Lights		
Engine Smoke			
Hatch			

Table 12. DIS Appearance Field Values

The DIS Entity State PDU also has the **Dead Reckoning Parameters** field. This field contains three fields, one of which is the **Dead Reckoning Algorithm**. This field allows a entity to inform other exercise participants what dead reckoning model to use when computing its new location and orientation (27:11).

3.3 Initial Class Structure Analysis

Using the SIMNET and DIS analysis as a guide, it was determined that the major entities were structures, munitions, and vehicles. The only difference between structures, munitions, and vehicles were Vehicle Appearance PDU fields that become useful for non-structures, such as velocity. Vehicles are decomposed into air, ground, and water vehicles, using the DIS **Entity Appearance** field. In addition, a tank vehicle, with the turret azimuth and gun elevation, was needed.

This arrangement was a candidate for class generalization and inheritance (34:38-41). Each lower level in the class hierarchy added information necessary for its own existence, such as gun elevation for tank. In addition, the lower level classes could inherit the and attributes operations from the higher level classes.

One of the major reasons for using this class structure is the ability to easily add attributes to the individual classes or the class structure as a whole. This is useful as AFIT continues distributed simulation research into the future and finds additional information that needs to kept on individual entities. Another reason is the ease of which another subclass can be added to the structure, like the munition classes.

A decision was made to not create any type of structure for a life form entity. A life form will be almost impossible to view from the virtual cockpit. The synthetic battlebridge can zoom in on a life form, but that provides no information to the user.

Figure 1 shows the entire entity class structure. The munitions class was put there to show its place when inserted into class structure. See Chapter V for a discussion on the future of the

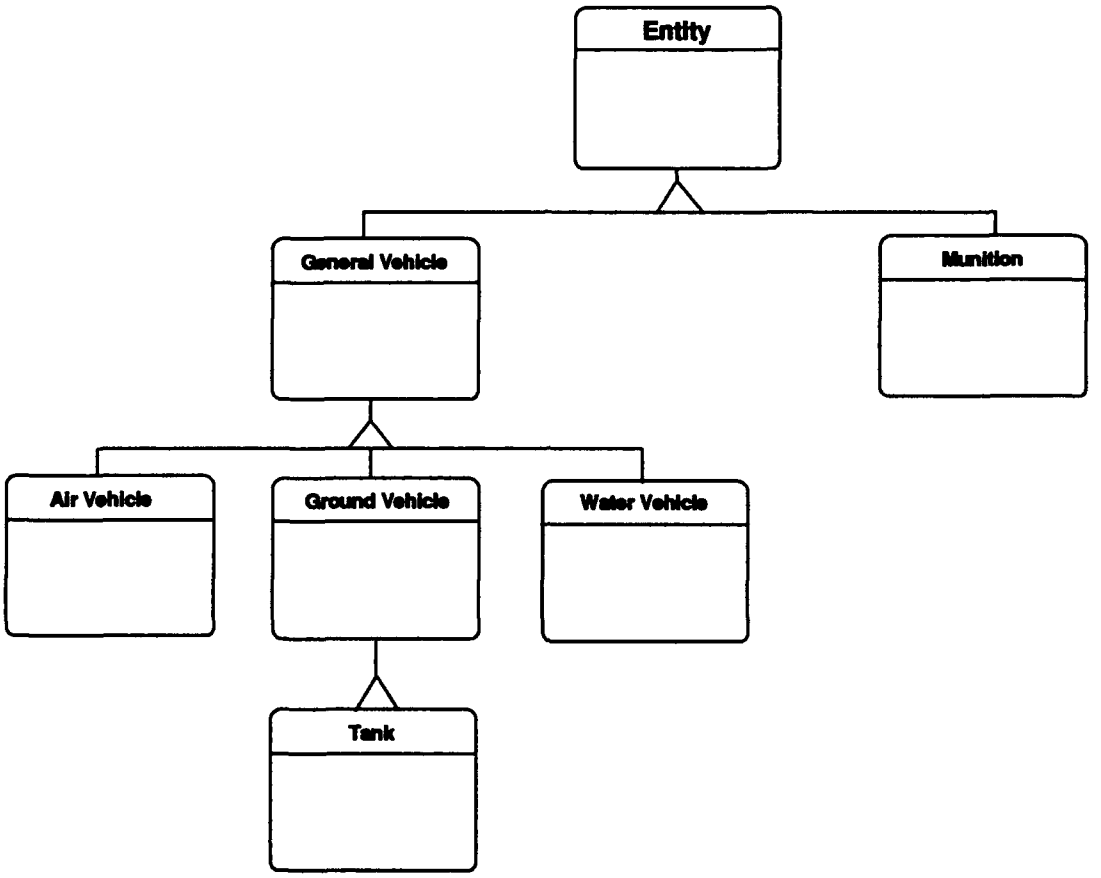

Figure 1. Entity Class Structure

munitions class.

3.4 Design

A record, similar to the format of the SIMNET `ObjectType`, was needed for use as a means of identifying the type of entity being simulated. The `entityType` type was created. The format for the record:

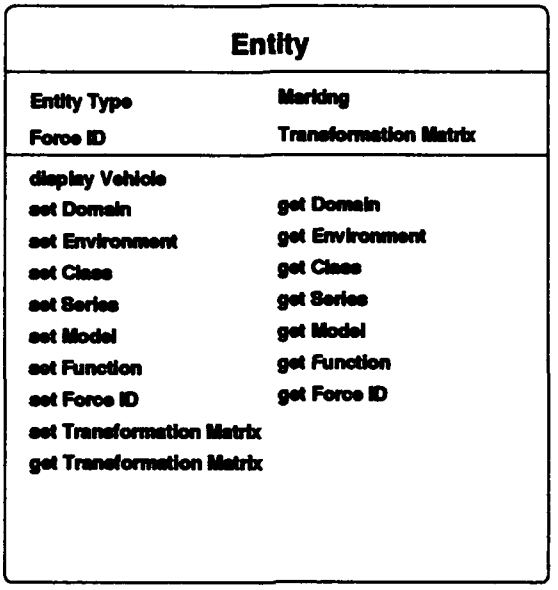

Figure 2. Entity Class

```
typedef struct
   entityDomain         Domain;
   entityEnvironment    Environment;
   entityClasses        Class;
   entityCountry        Country;
   entitySeries         Series;
   entityModel          Model;
   entityFunction       Function;
}
```

The first class created was the Entity class. This class serves as the root of the class hierarchy. In addition, this class will hold data on any static or stationary object. Figure 2 shows the generic entity class.

The only four attributes to the Entity Class are the Entity Type, Marking, Force ID, and the transformation matrix. All attributes are private, that is these attributes are only accessible within the class. All of the operations are public. These operations are the interfaces to the objects within the class (44:146).

One of other operation exists in Entity Class (as well as a all subclasses). A class constructor exists with the same name as the class name, in this case Entity Class. The constructor is used

to initialize the attributes in the class (44:30). In this case, the attributes are initialized to a set of zero values. This prevents any attribute not being set to a value if accessed later in program execution.

From the Entity Class, two subclasses were created: Vehicle Class and Munition Class. The status of the Munition Class is discussed in Chapter V. The Vehicle Class, being a subclass to the Entity Class, inherits all the attributes and operations from that class. From figure Figure 1, the class structure is an example of single inheritance, where each derived class has only one base class (23:395).

The Vehicle Class was designed to include all entities (excluding munitions) that can move through the SIMNET gaming area. The additional attributes in Vehicle Class are Capabilities, Linear Velocity, Linear Acceleration, Angular Velocity, and Dead Reckoning Algorithm. The Capabilities are a record in the same format for the SIMNET **VehicleCapabilities** record. Since **VehicleCapabilities** are only associated with moving vehicles, this is a logical spot to place Capabilities record. Once again, all attributes are private. The only way to access the information is through the operations. Figure 3 shows the Vehicle Class Structure.

From the Vehicle Class, three subclasses were created: Air Vehicle Class, Ground Vehicle Class, and Water Vehicle Class. Once again, these subclasses inherit the attributes and operations from the Vehicle Class. Even though SIMNET has an environment code for space, a Space Vehicle Class was not created. In the near future, it does not seem likely that space vehicles will participate in SIMNET exercise. The difference among these three subclasses is the Appearance field. The Appearance field is a record of the same fields as the DIS **Appearance** field. Figure 4 shows the Air Vehicle Class and Figure 5 shows the Ground Vehicle Class.

As stated in a previous section, the DIS **Appearance** field has separate definitions for above-surface ships and below-surface ships. These two **Appearance** fields were combined into one longer record for use in the Water Vehicle Class. Figure 6 shows the Water Vehicle Class.

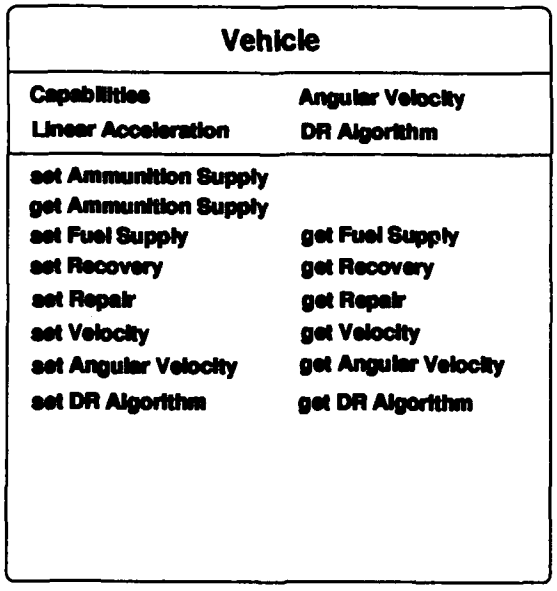

Figure 3. General Vehicle Class

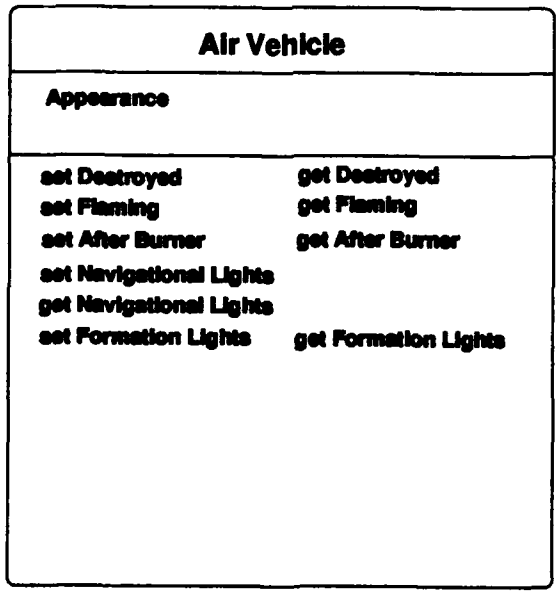

Figure 4. Air Vehicle Class

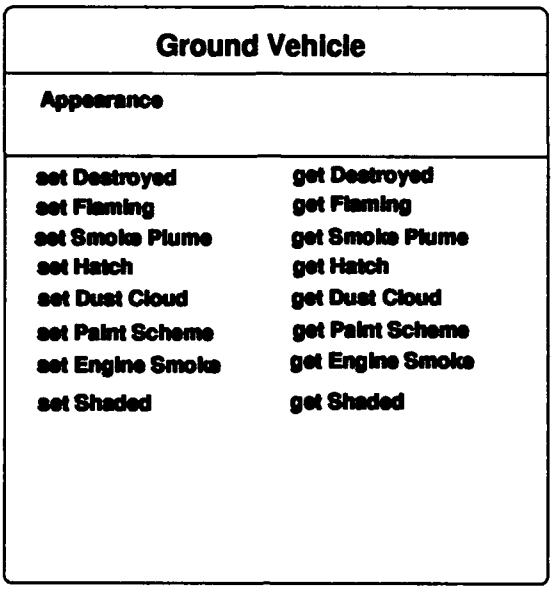

Figure 5. Ground Vehicle Class

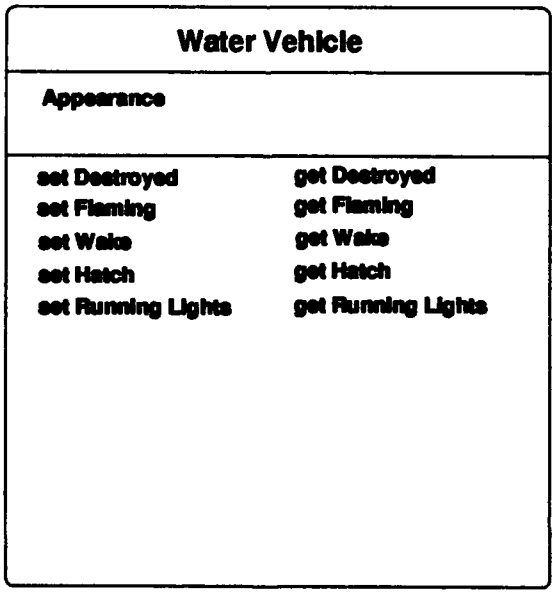

Figure 6. Water Vehicle Class

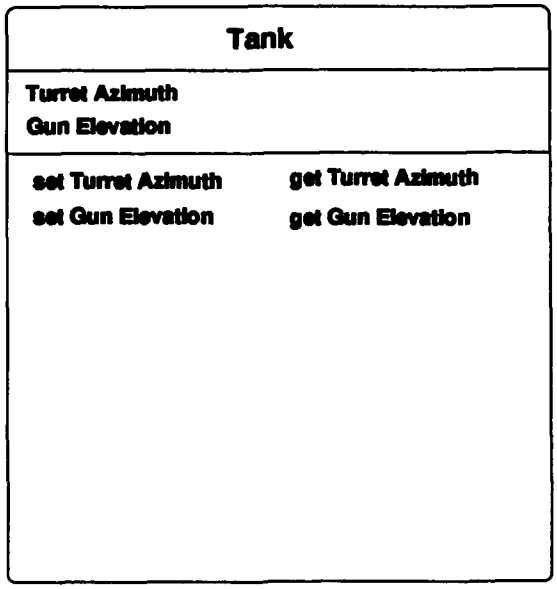

Figure 7. Tank Class

Finally, a Tank Class was added as a subclass to the Ground Vehicle Class. This Tank Class adds the two variables that used for SIMNET vehicles with independently moving guns and turrets: Turret Azimuth and Gun Elevation. While the original SIMNET definition only had in mind Army combat vehicles, War Breaker Exercise testing has proven otherwise. Vehicles such as the SCUD launcher are also coming across as SIMNET Tank Class vehicles. While a SCUD launcher does not have a gun or turret that moves, a SCUD launcher does have a cradle arm that moves to the vertical position to fire (19:111). Figure 7 shows the tank class.

3.5 Summary

The entity class structure was created in order to contain information about the different types of entities that are participating in a SIMNET exercise. These entity classes will be used by the entity object manager, discussed in the next section, for use during a simulation.

IV. Entity Object Manager

4.1 Introduction

Simulators that participate in distributed simulations are required to keep track of information about the other entities participating in a simulation. This was the reason for the creation of the Entity Object Manager. The Entity Object Manager performs a wide variety of necessary functions of a simulator including inserting and deleting vehicles from a simulation. This section explains in detail the Entity Object Manager used in the virtual cockpit and the synthetic battlebridge.

4.2 Analysis

4.2.1 Simulator Requirements As stated above, a simulator has a number of requirements to perform when participating in a SIMNET simulation. Even with the great volume of documents written on SIMNET and DIS, there does not exist a hard set of detailed requirements for simulators to follow. Most of the requirements are implied (like inserting a vehicle when a new vehicle enters the simulation) or inferred from the minute details of the Protocols (such as delete a vehicle that has not broadcast a SIMNET Vehicle Appearance PDU within the last twelve seconds).

There does seem to exist a set of important functions that all simulators must perform in order to properly participate in a SIMNET simulation. These functions are:

- Inserting vehicles into a simulation using the Activate Request PDU and Activate Response PDU. The most common scenario in which an Activate Request PDU is used is after one vehicle (vehicle A) has towed another vehicle (vehicle B) into its starting location. After Vehicle A is finished towing Vehicle B, Vehicle A issues an Activate Request PDU to Vehicle B to activate it. Vehicle B replies with an Activate Response PDU and a new vehicle is in motion (31:81-86).

- Inserting vehicles into a simulation using the Vehicle Appearance PDU. A new vehicle entering an exercise broadcasts its first Vehicle Appearance PDU. Other simulators receive the PDU and compare the `vehicleID` to others already being tracked. If it is a new `vehicleID`, this new vehicle is inserted (31:88).

- Deleting vehicles from a simulation using the Deactivate Request PDU and Deactivate Response PDU. In a manner similar to the activate process, a vehicle (vehicle A) issues a Deactivate Request PDU to another vehicle (vehicle B). Vehicle A has told Vehicle B to shut down. Vehicle B replies with a Deactivate Response PDU (31:86-88). The Deactivate Response PDU does not exist in the WAR BREAKER SIMNET formats (37).

- Deleting vehicles from a simulation from lack of Vehicle Appearance PDU transmissions. If a simulator does not broadcast Vehicle Appearance PDUs at least twelve seconds apart, the object is deleted by all other simulators (31:88).

- Updating information on a vehicle in a simulation. The Vehicle Appearance PDU is used in this case. A simulator will issue a Vehicle Appearance PDU at least every 5 seconds while it is active in the exercise. This Vehicle Appearance PDU will contain updated location, orientation, velocity, and status for the vehicle in question (31:88-92).

- Dead reckoning of moving objects in a simulation. As discussed in Chapter II, dead reckoning is used to reduce to network bandwidth. Therefore, a function of any object manager is to dead reckon any moving vehicles. Different dead reckoning algorithms can be used depending on the vehicle type (31:22-23). A tank requires only the simple linear dead reckoning model, while aircraft may require the complex harmonic dead reckoning model.

4.2.2 Network Requirements Since the Entity Object Manager has the responsibility of managing the various entities that participate in an exercise, the Entity Object Manager must interface to the network communications modules to retrieve SIMNET PDUs from the network. As stated in Chapter I, two network communications packages are used by the Entity Object Manager to receive Vehicle Appearance PDUs.

The first network communications package was developed by Mr John Locke at NPS. This software was developed to run on both IRIS workstations and Sun workstations (24). Unfortunately, all applications using this network harness must run at root privileges since the network harness transmits and receives raw Ethernet packets. This is a necessary requirement, however, since other SIMNET simulators only transmit and receive raw Ethernet packets.

The other network communications package was developed locally by Mr Bruce Clay. The package was developed with the intent of allowing students to continue developing and testing the software without using root privileges. This package also transmits and receives SIMNET PDUs

like the NPS software. However, an additional UDP header is used for broadcasting. This allows application programs to run at normal user privileges. Appendix A has the details on the AFIT network communications software.

4.2.3 Virtual Cockpit Requirements Since the virtual cockpit involved a number of thesis students, software integration was a possible problem. Capt McCarty's software was used to render all objects. Therefore, the Entity Object Manager needed an operation that could be called by the rendering software to get information on the various entities involved in the simulation.

Capt McCarty's rendering software only needed the type of vehicle and its transformation matrix in order to properly render an object. This structure, **entityData**, holds the information required on a single entity.

```
typedef struct {
   int              ID;
   Matrix           TMatrix;
   entitySeries     EntityType;
} entityData;
```

The ID is the Entity Object Manager identifier for that entity. The 4 x 4 transformation matrix and the series of the entity (such as F-15 and SCUD launcher) are also passed to the rendering software.

In order for the virtual cockpit to be seen by other entities, the Entity Object Manager also transmits Vehicle Appearance PDUs (which is a WAR BREAKER Vehicle Appearance PDU) out onto the network. An interface from Capt Switzer's flight dynamics classes passes the necessary information needed to populate the Vehicle Appearance PDU. Table 14 lists the Vehicle Appearance PDU fields that are updated before broadcasting over the network. Most of the Vehicle Appearance PDU will be same for each transmission. Table 13 lists some of the Vehicle Appearance PDU fields and their permanent values.

Field	*Value*
Vehicle ID - Site	AFIT
Vehicle ID - Vehicle	1
Vehicle Class	Simple
Force ID	0
Vehicle Guises	F-15E
Vehicle Markings	None

Table 13. Virtual Cockpit Vehicle Appearance PDU Static Field Items

Current Orientation Matrix
Linear Velocity in X, Y, and Z
Linear Acceleration Vector in X, Y, and Z
Angular Velocity about the X, Y, and Z-axis

Table 14. Virtual Cockpit Vehicle Appearance PDU Dynamic Field Items

4.2.4 Synthetic Battlebridge Requirements Since the synthetic battlebridge also involves two separate thesis efforts, software integration was a possible problem. Like the virtual cockpit, the Entity Object Manager needed an operation that could be called by the synthetic battlebridge to obtain information on the simulation entities.

Capt Haddix's software needed more information than Capt McCarty's so another structure was needed to hold information about a single entity. This structure, `object`, is

```
typedef struct {
   int                object_id;
   boolean            active;
   entityCountry      country;
   entityEnvironment  obj_type;
   entitySeries       obj_series;
   Matrix             position;
   boolean            display;
   boolean            selected;
   boolean            threat;
   boolean            radiating;
   boolean            friendly;
} object;
```

Like the virtual cockpit rendering software, the synthetic battlebridge requires the entity

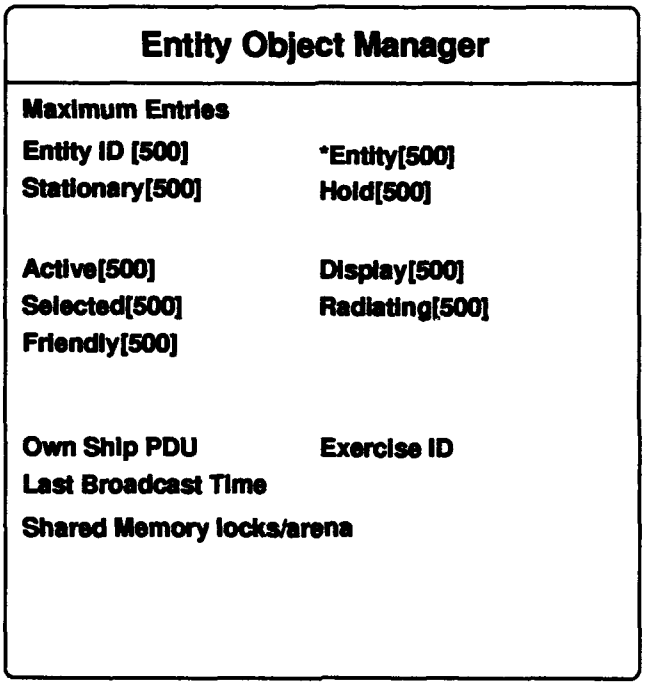

Figure 8. Entity Object Manager Class (Attributes)

identifier, the entity's transformation matrix, and the entity's series. In addition, the synthetic battlebridge uses information such as the entity's country and the entity's environment (such as air or ground). The active, display, selected, threat, radiating, and friendly boolean flags are used by exclusively by the synthetic battlebridge. However, these flags are part of the record maintained on individual entities.

4.3 Design

The Entity Object Manager was developed as a single C++ class. Figure 8 shows the Entity Object Manager Class and its attributes. Figure 9 shows the Entity Object Manager Class and its methods.

One of the first decisions made was to limit the number of vehicles that the Entity Object Manager can track to 500. SIMNET simulators should be able to handle at least 500 objects (31:2). The sights were set on this minimum goal.

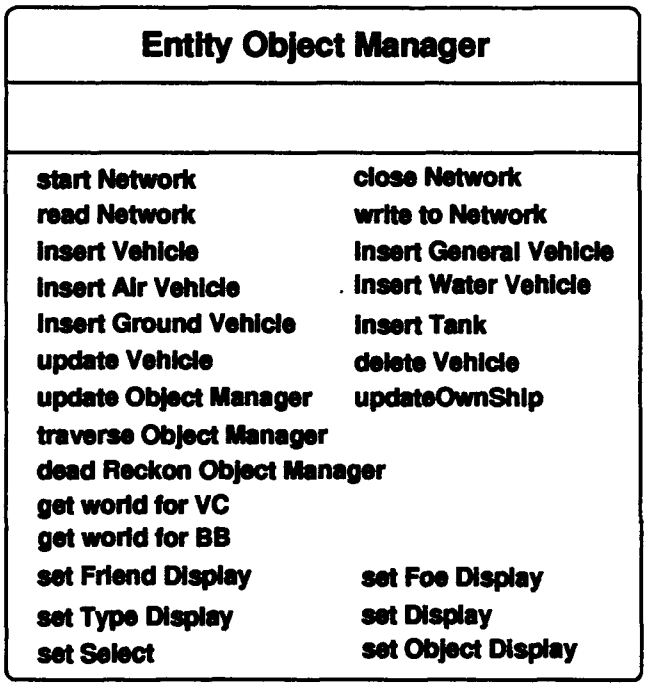

Figure 9. Entity Object Manager Class (Operations)

The Entity Object Manager is set up internally as a series of 500 element arrays. The ID of the array element is the Entity Object Manager internal identifier for that entity. This ID is also passed to both the virtual cockpit rendering software and the synthetic battlebridge.

Some of the attributes in the Entity Object Manager are used to track information on a particular entity. The **Entity ID** is the three part SIMNET `VehicleID` type. The **Stationary** flag is used to determine whether an entity is moving or not moving. This is used to determine which entities to dead reckon. If an entity is not moving, there is no need to dead reckon. The ***Entity** is the pointer to the appropriate class structure.

The attributes **Hold**, **Active**, **Display**, **Selected**, **Radiating**, and **Friendly** are used only for the synthetic battlebridge. Table 15 lists the data that describes an entity.

Some of the other attributes are used in broadcasting our current status using a Vehicle Appearance PDU and thus only used by the virtual cockpit. The **Own Ship PDU** is the Vehicle Appearance PDU on the virtual cockpit. The information in the individual fields are modified

Entity ID - Site
Entity ID - Host
Entity ID - Vehicle
Stationary Flag
Hold Flag
Active Flag
Display Flag
Selected Flag
Radiating Flag
Friendly Flag
Pointer to Appropriate Class

Table 15. Entity Attributes

by the appropriate get and set operations. The **Exercise ID** is the current exercise that the virtual cockpit is participating. This number is an integer. The **Last Broadcast Time** is the last simulation time that a Vehicle Appearance PDU was broadcast by the Entity Object Manager. This field is used for dead reckoning.

The remaining attributes are shared memory locks and arenas. These are used when the applications are in multiprocessor mode.

As with the entity class structure, none of the attributes are public. They are only accessible through the operations of the Entity Object Manager.

The operations on the Entity Object Manager are divided into both public and private operations. The public operations are listed in Table 16. All other operations are private.

The Entity Object Manager class constructor performs a series of housekeeping functions. All array pointers are set to null values. All other values are initialized to a default state. The constructor also starts the network interface by calling the **start Network** operation.

The **update Object Manager** is the workhorse operation in the object manager. This operation calls the operation that reads SIMNET PDUs from the network, **read Network**. Once a PDU is received, the function determines whether this entity already exists in an array element.

update Own Ship
traverse Object Manager
close Network
get World for VC
get World for BB
set Friend Display
set Foe Display
set Object Display
set Display
set Type Display
set Select

Table 16. Entity Object Manager Public Operations

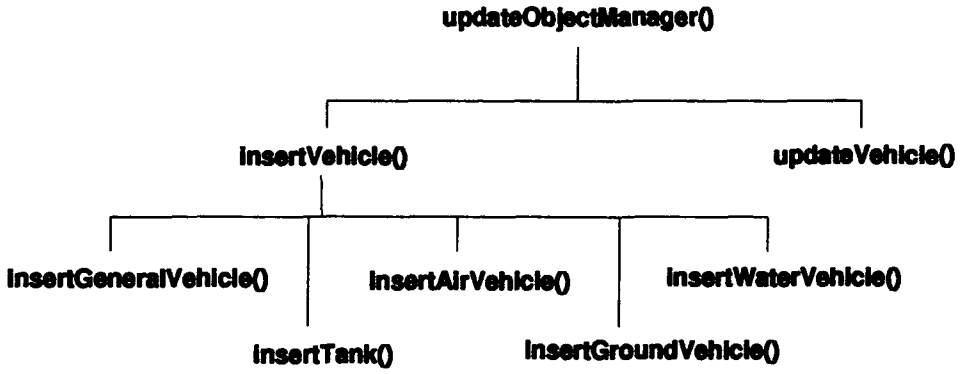

Figure 10. Entity Object Manager Insert/Update Operations Hierarchy

If it does, it passes the PDU to the **update Vehicle** function to update the location, orientation, velocity, and acceleration of the entity.

If the entity is a new vehicle to the Entity Object Manager, **update Object Manager** passes the Vehicle Appearance PDU to the operation **insert Vehicle**. This function parses the first series of bits of the **guises** field in the Vehicle Appearance PDU. That parsing determines the environment of the vehicle. The Vehicle Appearance PDU is then passed to the appropriate operation for final parsing of the Vehicle Appearance PDU and insertion of a new class object into an empty array element. Figure 10 shows the functional hierarchy among the various update and insert functions. Only **update Object Manager** is visible outside of the Entity Object Manager.

The **deadReckonObjectManager** traverses the array and finds any non-stationary vehicles. These vehicles will have their location updated after execution of this process following the appropriate dead reckoning algorithm.

The **updateOwnShip** operation is the interface from Capt Switzer's flight dynamics modules. **updateOwnShip** then calls writeToNetwork to broadcast the new Vehicle Appearance PDU into the exercise.

The interface to Capt McCarty's rendering modules in the virtual cockpit is the function **get_World**. This procedure passes a pointer to an array of all vehicles known to the entity object manager. This array is generated in the **get_World** function.

The interfaces to Capt Haddix's Synthetic Battle Bridge are the following functions:

- **get_world** - This procedure produces a pointer to an array of all vehicles known to the entity object manager. This array is generated in the **get_world** function.
- **set_friend_display** - This procedure sets the **Friendly** display variable for the display in battle bridge.
- **set_foe_display** - This procedure also sets the **Friendly** display variable. The **set_foe_display** performs the opposite function of the **set_friend_display**.
- **set_object_display** - This procedure sets an individual element **Display** variable.
- **set_display** - This procedure sets the **Display** variable for all vehicles.
- **set_type_display** - This procedure sets the **Display** variable for vehicle of a certain type.
- **set_select** - This procedure sets the **Selected** variable for an individual vehicle.

4.3.1 Design Considerations The Entity Object Manager was originally developed to have a simple structure that would working as soon as possible. A fixed length element array structure is very simple yet inefficient as the number of vehicles approach the limit. The update algorithm starts at the first element and traverses the array until it reaches the desired element or all filled

elements are searched. The insert algorithm finds the first empty element to insert a new entity. The algorithms are inefficient when designed. The choice was made to allow for a simple and working solution to the problem. The problem would be solved by spawning the object manager off to another processor. With the power of one processor in the current line of IRIS workstations, the processors should overcome any inefficiencies in the Entity Object Manager.

4.4 Summary

The entity object manager is the means by which the virtual cockpit and the synthetic battle bridge keep track of data on the other participants in a SIMNET exercise. The entity object manager performs the functions necessary for SIMNET simulations, such as dead reckoning.

V. Results and Recommendations

5.1 Conclusion

The problem statement in Chapter 1 identified the need for an entity object manager that kept track of other objects participating in a SIMNET exercise. The creation of the entity class structure and the entity object manager provides a solution to that problem.

The entity class structure provides a foundation for tracking individual data items for the various entities involved in a simulation. In addition, the entity class structure can be modified easily to accommodate new information in the future. Additional, classes can be added to the structure to take into account more types of entities, such as space entities.

The entity object manager provides the functionality required to manage SIMNET entities during a SIMNET exercise. The basic functionality is present: insertion, deletion, updating, and dead reckoning. The entity object manager also serves as the virtual cockpit and synthetic battlebridge's gateway to the rest of the SIMNET network. In the future, this entity object manager will by the gateway to the DIS network.

The Entity Object Manager has been in operation for over two months. The initial interface problems between the other modules of the virtual cockpit and the synthetic battlebridge have been corrected. While the network communications package is sometimes unstable at times due to high network traffic, the Entity Object Manager works well with both network communications packages.

Recent tests have proven that the Entity Object Manager is functioning. Recent trips to the Institute for Defense Analyses (IDA) Simulation Center have proven that the virtual cockpit can communicate with other entities using the Entity Object Manager. Other simulators saw the virtual cockpit flying the gaming area as an F-15E. The virtual cockpit was rendering other objects participating in the exercises.

A recent test with NPS proved the functionality of the systems over a long-haul communications line. The synthetic battlebridge displayed both local SIMNET message traffic and SIMNET message traffic from NPS. The virtual cockpit was flying well both at AFIT and NPS. The Entity Object Manager has been proven to work.

5.2 Recommendations for Future Research

The first recommendation is the migration to the DIS standard. At some future date, defense related simulations might migrate to the DIS standard. This date was originally November 1992, but it is has been slipped. The NPS network communications software as well as various portions of the object manager will have to be modified for a DIS simulation. The entity class structure may also need minor modifications.

One possible problem is the inefficient algorithms in the entity object manager. There has not been enough vehicles on the network to perform a Load/Stress test on the object manager (20:202). In the long run, it may be necessary that the object manager be modified with more efficient insertion and search algorithms. This is especially true when porting applications to single processor workstations.

Another recommendation is the munitions class structures. An initial attempt has been made to create the munition class structure. There are four munitions subclasses defined by SIMNET (31:151-157):

- Ammunition, such as high explosives and proximity bombs,
- Bombs, such as those dropped from aircraft,
- Mines,
- Missiles,

There are some questions about what information needs to be kept on these munitions. For instance, it does not make sense to keep track of mines, especially from the point of view of the synthetic

cockpit since the virtual cockpit can not render mines. However, the mine explosion underneath a ground vehicle is important.

Events are not yet handled by the entity object manager. Movement of vehicles is considered an event, however Vehicle Appearance PDU transmissions take care of that. Other events include the firing and subsequent detonation of a munition and collision of objects. The entity object manager may require modifications to handle event information from the various PDUs.

5.3 Summary

The entity object manager and entity object classes are operational for future AFIT distributed simulation and synthetic environments research. AFIT now has the capability to transmit, receive, and process distributed interactive simulations messages. The concentration can now be directed at pushing the technological edge further that it has ever gone at AFIT.

Appendix A. *AFIT Distributed Simulation Network Broadcast Software*

A.1 Preface

This appendix is taken from a users guide file written by Mr Bruce Clay to explain the details of the AFIT developed network communications package.

A.2 Introduction

This document describes a software package to be used with network distributed interactive simulation applications. The program uses standard UNIX shared memory and semaphore calls to simplify message passing from one host to another or multiple processes on the same host. The current version of the package sends messages to remote hosts using UDP/IP subnet broadcasting on a port effectively specified by the user. Simulation groups are defined which relate to individual ports and are used to control which simulation processes receive the message. The first port of each network daemon process is used as a control port between the client program and the network daemon using the loopback address instead of the broadcast address. All other ports requested when the daemon is started are used for group communications. For every port requested at startup, a shared memory block and a semaphore are allocated. All procedures and their parameters are described in the following sections.

A.3 Program Operation

The maximum number of semaphores allowed on a single UNIX machine is set by the operating system to 50. However, Silicon Graphics imposes an additional restriction that a single process may only use 25. The first semaphore of each daemon process started is used to hold the number of buffers allocated when the daemon was started. That leaves 24 semaphores available to each of 2 daemons if the daemons are started with maximum allocation. The number of buffers allocated are set by the command line option **-b <number of buffers>**. Under the current configuration

up to 4 daemons can be used at any given time. To start a daemon using other than the default values use the option **-d <daemon number 0 - 3>**. The daemon number used also sets the base network port to be used so the client program must use the same daemon number in all of the calls requiring a daemon value. The client program can talk to multiple daemons if desired as long as an **Init_client** call is made for each one. To stop the daemon process, invoke the daemon again with a **-q** command line option. If multiple daemons have been started, the **-q** option must be used with the **-d** option and must appear on the command line after the **-d** option. *REMEMBER* - other people may be using the daemon so check with other users before stopping any daemon. By default all of the semaphores and shared memory are allocated with read and write permission to everyone. This can be changed by the client if group access control to a given port is desired. The daemon has been complied with a maximum network buffer size of 512 bytes. This can be changed by changing **DEFAULT_SIZE** in **simnet.h** and recompiling the package.

In the client program, four parameters have been defined to set which daemon to talk to (**DAEMON1, DAEMON2, DAEMON3, DAEMON4**). Groups can be set up the same way using a value between 1 and 23 to set the port offset (such as **A_FLT, B_FLT, RADAR_1** etc). Although the messages are broadcast to all the machines on the subnet, they will only be received by daemons listening the the respective ports. Shared memory and semaphore access permissions can be changed in the client in the second parameter to **Init_client**. These permissions are set via an octal value the same way UNIX file permissions are set. See the man page for **chmod** for more information.

A.4 Program Description

A.4.1 Client sample program: **simnetc.c** The client program must be initialized by calling **Init_client** before actually sending any messages. Any time the client wants to send a message, locate an unused buffer by calling **Get_free_buf**. If an unused buffer is available, the preformatted message can be copied to shared memory using **Write_buf**. Multiple messages can be written to

shared memory before any of them are sent. When all desired messages have been written in shared memory, tell the daemon to send them to the remote processes by calling **Notify_daemon**. When the client program wants to receive a message, it should call **Read_buf**. This must be done periodically even if the client is not ready to use the incoming message, otherwise the daemon will run out of free buffers and messages may be lost. In addition, the client will not be able to send any more messages. If the client is not ready, it should just copy the message into local memory until it is ready to use the data. Messages sent by the daemon are also received back as an echo to be used by other processes on the same machine. If these messages are not desired, the client should set the variable **ignore_local** to **TRUE** before calling **Read_buf**. The last parameter used by **Read_buf** tells the program how to set the semaphore attached to the message buffer. If the parameter is set to **SEM_CLEAR** the buffer will immediately be freed for use. If the parameter is **SEM_MARK** the buffer will not be used unless there are no free buffers. This can be useful if other processes on the same machine may need to use the data in this message buffer.

When compiling the main block of client code, **#define MAIN_CODE** should be inserted before the **#include "simnet.h"** line to satisfy variables required in the main module that are used by the library functions. Most of the modules in this package have debugging statements built in that can be displayed by including the line **#define DEBUG_ON**. Other information that monitors progress on the different routines can be displayed by including **#define VERBOSE**.

A.4.2 Daemon program: **simnetd.c** The server program is responsible for allocating the shared memory and the semaphores as well as initializing the network used by this package. A network port is initialized for each buffer that was requested when the daemon was started but the buffers are not explicitly tied to the respective port. Rather, the buffers are set up as a pool to be used by any port for network message passing. To keep the daemon from using excessive CPU cycles, interrupts have been implemented for all network traffic on all platforms except the Intel Hypercube. The operating system on the Hypercube does not support interrupts for the

network in the same way as the other systems, if at all. After all of the necessary initialization has been completed, the daemon enters the main program loop. If compiled for polled mode, the semaphores are continually monitored in the main loop. The Hypercube requires polled mode. If polled mode is not required, control flags that are set by the interrupt procedure are checked. If the appropriate control flags are found set, specific operations are performed based on the flags values. When all flag controlled operations are completed, the network ports are checked to see if any data came in while performing other operations. If no new data has arrived, the program pauses until an interrupt wakes it up again and the cycle starts over again. Since user modification of the daemon is not expected, the procedure call sequence is not defined in this document. All of the code is documented and the library calls used by the daemon are defined in the following sections. Instructions to recompiling both the daemon and sample client programs are also discussed in the next section.

The conditional compile statements **MAIN_CODE**, **DEBUG_ON** and **VERBOSE** are described in the **simnetc.c** section above. The package has been compiled and tested on SUN Sparcs, Silicon Graphics IRIS 4D, and Intel IPSC II. The **makefile** included with this package is set up to work with all three of these systems. To compile the library type **make (sgilib** or **sunlib** or **ipsclib)**. To compile the daemon type **make (sgid** or **sund** or **ipscd)**. To compile the sample client program type **make (sgc** or **sunc** or **ipscc)**.

A.5 Library Description

The library function descriptions used with this package are described below followed by the description for the daemon and a sample client program.

A.5.1 **Bytes_ready**

```
int Bytes_ready(fdes) - procedure in file b_ready.c
int fdes;
```

This procedure returns the number of bytes available to be read from the socket file descriptor **fdes**.

A.5.2 Check_all_ports

Check_all_ports() - procedure in file checkall.c

This procedure sets the **control_mess** flag if a control message has been received on the control port and the **data_mess** flag if a message has been received on any data port. This procedure is used in the daemon and called by the **SIGIO** interrupt procedure and right before the daemon goes to sleep.

A.5.3 Check_overflow

int Check_overflow() - procedure in file chk_over.c

This procedure first looks for a buffer whose associated semaphore is set to **SEM_CLEAR** and if not found looks for one set to **SEM_MARK**. If no buffers could be found, the control semaphore is set to indicate an overflow condition (**SEM_OVER**) and a **TRUE** is returned to the calling procedure. Otherwise, a **FALSE** is returned to indicate that there is not an overflow condition. The procedure is called in the daemon control loop when a data message has been received and when a message has been sent on the network to insure buffers are not over written.

A.5.4 Find_buffer

```
int Find_buffer(mode) - procedure in file find_buf.c
int mode;
```

This procedure looks at the semaphores to find a buffer that is set to **mode** and returns the buffer number. The procedure is called in **Check_overflow** and **Recv_buf** in the daemon but can be useful for client programs also.

A.5.5 Get_free_buf

```
int Get_free_buf(daemon, timeout) -  procedure in file get_free.c
int daemon, timeout;
```

This procedure is used only by the client program. The **daemon** value is tested to be within the proper range and not **in_use**. Then the total number of buffers is found and all semaphores are checked until one is found in the **SEM_CLEAR** state. If none are clear, the buffers are rechecked to see if any are in the **SEM_MARK** state. If any buffers were found, the semaphore is set to **SEM_HOLD** to reserve it for later use and the buffer number is returned to the calling procedure. Otherwise, **-1** is returned to the calling procedure. The **timeout** parameter can be used to make the procedure look for an available buffer multiple times if none are available. When data is sent through the network, the daemon clears the semaphore so buffers may be freed while in this process loop. Refer to the Section A.3 for more details.

A.5.6 Get_port

```
int Get_port(port, send_mode) -  procedure in file get_port.c
int port, send_mode;
```

This procedure is useful *only* in the daemon program. It is used to get a socket file descriptor for **port** and configure the port for interrupts. The **send_mode** parameter and be an **OR**'ed combination of **SEND** and **RECV**. Only one program can be setup to **RECV** on a given port. See note 3 in Section A.6 of this document for more details.

A.5.7 Get_shared_buf

```
int Get_shared_buf(daemon, permission) -  in file get_shar.c
int daemon, permission;
```

This procedure is used to locate the shared memory allocated by the daemon. The **daemon** parameter is verified to be within range and not in use. The semaphore ID (**semid**) and shared memory ID (**shmd**) are located based on **key** values predefined in **simnet.h**. These key values were

selected to double as network port values with a separation of 25 based on the Silicon Graphics maximum number of semaphores per process. The number must also be set high enough to be in the range of ports that are not reserved by the operating system. If both IDs were obtained, the **in_use** flag is set for use by other procedures to keep track of the total number of daemons currently being used. The **permission** value should be any octal number in the same manner as used with files permissions. See the man page for **chmod** for more details. The procedure is required in the client program before calling **Read_buf** and is called by **Init_client** in the sample client program.

A.5.8 Get_broadcast_addr

```
Get_broadcast_addr(addr_str) -  procedure in file getbroad.c
char _addr_str;
```

This procedure finds the local hosts IP address and changes the last number to a **0**. This address is then used for the IP subnet broadcast address.

A.5.9 Init_client

```
Init_client(daemon, permission) -  procedure in file init_cli.c
int daemon, permission;
```

This procedure is only used in the client program. The procedure calls **Get_shared_buf** to locate the shared memory and semaphore ID values, then calls **Get_port** to get a file descriptor to use with **Notify_daemon**. The actual interrupt procedure **Set_recv** is also contained in this file but is blocked out by conditional compile statements until interrupt problems are resolved. See note 3 in the section A.6 for more details.

A.5.10 Notify_daemon

```
Notify_daemon(fdes, command) -  procedure in file notify_d.c
int fdes, command;
```

This procedure sends a network loopback message to the daemon on the control port. The **fdes** parameter is the file descriptor returned from the call to **Get_port**. The **command** parameter is reserved for future use and should be set to **SEM_SEND** in current client programs. The procedure is used in the client program to wakeup the daemon and in the daemon program when the daemon is invoked to shutdown another daemon process.

A.5.11 Read_buf

```
int Read_buf(daemon, data_buf, sem_mode) -  procedure in file read_buf.c
int daemon, sem_mode;
struct mess_rec _data_buf;
```

This procedure is used in the client only. First the **daemon** value is verified to be within range and not **in_use**, then the procedure attaches to the shared memory segment. The procedure checks the **TOT_BUF_SEM** to find out how many buffers this daemon has and loops through the buffer-semaphore pair until it finds one with a semaphore set to **SEM_RECV** or **SEM_LOCAL**. If the client program set **ignore_local** to **TRUE**, the respective semaphore is changed to **SEM_CLEAR** and the data is ignored. Otherwise, the data is treated as if the message came in from a remote site. This is to allow multiple client programs on the same host to function as if they were on remote systems. If either a remote message or a desired local message was received, the data is copied to the local buffer specified by **data_buf**. The semaphore is then set to the mode specified by **sem_mode** which should be either **SEM_CLEAR** or **SEM_MARK** but could be set to **SEM_RECV** if the client program wants to reread the message for any reason.

REMEMBER - there are a fixed number of buffers to be used by the client and daemon and only buffers in the **SEM_CLEAR** or **SEM_MARK** state will be reused. The **SEM_MARK** state is used when there are multiple processes on the same machine that may want to read the same message. When all buffers have been checked the procedure detaches from the shared memory segment.

A.5.12 Recv_buf

```
int Recv_buf(group, mess_buf) -  procedure in file recv_buf.c
int group;
struct mess_rec _mess_buf;
```

This procedure is used only by the daemon. It first looks for a free (**SEM_CLEAR**) buffer. If none are found, the procedure will look for one that has been read but not set to clear (**SEM_MARK**). If there is an available buffer, the message is read from the network into the buffer and the associated semaphore is set to **SEM_RECV**. If no daemon buffers are available, the data is left if the systems network buffer (up to the system limit). The user must be careful to copy the data out of the daemons shared memory as soon as possible to prevent any data loss. This procedure is called in response to a flag set by the interrupt handler rather than being an interrupt service routine to keep interrupt time at a minimum. The **group** parameter is used to specify the network port offset from the base port used by this daemon. The **mess_buf** parameter is a pointer to the shared memory buffer the the daemon is to copy the data from the network into.

A.5.13 Send_all_bufs

```
Send_all_bufs(semid, mess_buf) -  procedure in file send_all.c
int semid;
struct mess_rec _mess_buf;
```

This procedure is used only by the daemon. It first finds the total number of buffers allocated to the procedure, then checks the semaphore for each buffer and if set to **SEM_SEND** calls **Send_buf**. The **semid** parameter is the semaphore ID returned by a call to **semget**. The **mess_buf** parameter is a pointer to the shared memory buffer with data to be sent out the network.

A.5.14 Send_buf

```
int Send_buf(semaphore, mess_buf) -  procedure in file send_buf.c
int semaphore;
struct mess_rec _mess_buf;
```

This procedure is used only by the daemon. The required network initializations are setup first then the message is sent through the port specified by the **mess_buf->group** parameter. If the data was successfully sent **semaphore** is set to **SEM_CLEAR**.

A.5.15 Write_buf

```
int Write_buf(daemon, buf_num, group, data_buf, data_len)
int semid, buf_num, group, data_len;      - procedure in file writebuf.c
char _data_buf;
```

This procedure is used used only by the client. The program first verifies the **daemon** value is valid (within range and not marked in use). The program then verifies that **group** is within one of the total number of ports based on number of port-buffer-semaphore groups allocated when the daemon was started. If **group** and **buf_num** are valid, the semaphore is set to **SEM_LOAD**. This is mainly for debugging purposes in case loading the message into shared memory fails. The client then attaches to the shared memory buffer, copies the data from the local buffer into the shared memory buffer then finally detaches from the shared memory. If all of this succeeds, the semaphore is set to **SEM_SEND**. If the client wishes to send more data at this time, **Write_buf** is called again with a new buffer number. If not, the client should call **Notify_daemon**.

A.6 Problems

Several problems were encountered while designing this package due to system differences. Hopefully, these differences can be resolved with future operating systems upgrade, which will be defined in this section.

1. Ideally, UNIX user interrupts could be used to allow the client to notify the daemon there is a message to be sent on the network. This could not be implemented because **SIGUSER1** interrupts can only be sent to "other processes of the same session" which turned out to be a given login session. Since other users on the same machine may want to send a message through the daemon,

the "same session" rule is broken. The work around this problem was to send a network message using the loopback address so other machines would not receive the message.

2. The DIS specification calls for IP network multicasting. The only operating system tested that currently supports IP multicasting is the Silicon Graphics' IRIX. To make all systems code compatible, IP/UDP subnet broadcasting was implemented. When the other system operating systems implement IP multicasting, the network portion for the daemon should be modified. When this can be done, simulation groups will be controlled by the multicast address used instead of the network port.

3. By using an `ioctl` call with `SO_REUSEADDR`, a socket is suppose to allow multiple processes to bind to the same socket. This did not appear to work correctly. For this reason, the control port is used in only one direction (from client to server). When this problem is fixed, the control port can be bidirectional so the client can be notified when a message has been received on the network. Until then, the client will have to poll the semaphores to find out.

Bibliography

1. Anderson, Gordon and Steve Seidensticker. "Networked Simulators: Using Models and Experience for Design." University of Central Florida Institute for Simulation and Training Library Reference Number B 04-b.
2. Bassiouni, M. *Simulation Networks Modeling and Monito g Final Project Report.* Prepared by University of Central Florida Institute for Simulation and Training for PM TRADE. Contract Number N61339-88-G-0002 Order 00008. 31 July 1989.
3. Bassiouni, M, Micheal Georgiopoulus and Jack Thompson. *Real Time Simulation Networking: Network Modeling and Protocol Alternatives.* Prepared by University of Central Florida Institute for Simulation and Training. Publication Number IST-TR-90-19. 1 November 1989.
4. Bassiouni, M, M. Georgiopoulus and J. Thompson. *Performance Evaluation of Local Area Networks for Real-time Simulation.* Prepared by University of Central Florida Institute for Simulation and Training. Contract Number N61339-89-C-0043. 6 February 1990.
5. Boner, K. E., D. R. Hardy and T. R. Tiernan. *Battle Force Inport Training/Simulator Networking: SIMNET Protocol Analysis.* Technical Note 1613, Naval Ocean Systems Center, San Diego CA. June 1990.
6. Boner, K. E., D. R. Hardy and T. R. Tiernan. *Battle Force Inport Training/Simulator Networking: SIMNET Protocol Suitability Considerations.* Technical Note 1614, Naval Ocean Systems Center, San Diego CA. June 1990.
7. Bouwens, Christina L. "Dead Reckoning Methods for Distributed Interactive Simulations." Prepared for PM TRADE by University of Central Florida Institute for Simulation and Training. April 1992.
8. Chung, James, et al. *SIMNET Simulation Description and Verification.* Bolt Beranek and Newman Inc Report Number 5855 prepared for Defense Advanced Research Projects Agency, December 1984.
9. *Draft Communication Architecture for Distributed Interactive Simulation.* Prepared for PM TRADE by University of Central Florida Institute for Simulation and Training, 1991.
10. *Draft Distributed Interactive Simulation Operational Concept.* Prepared for PM TRADE by University of Central Florida Institute for Simulation and Training, February 1992.
11. *Draft Distributed Interactive Simulation Standards Development Guidance Document.* Prepared for PM TRADE by University of Central Florida Institute for Simulation and Training, February 1992.
12. Foley, James D., et al. *Computer Graphics Principles and Practice.* Reading MA: Addison-Wesley, 1990.
13. Georgiopoulos, Michael. *Simulation Networking Protocol Alternatives.* Contract Number N61339-88-G-002 Order 0008. 31 July 1989.
14. Gerken, Mark J. 1991. *An Event Driven State Based Interface for Synthetic Environments.* MS thesis. AFIT/GCS/ENG/91D-07. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991. (AD-A243765).
15. Goel, Suresh C. and Kenneth D. Morris. "Techniques for Extrapolation, Delay Compensation, and Smoothing with Preliminary Results and an Evalation Tool." University of Central Florida Institute for Simulation and Training Library Copy. 20 September 1991.

16. Harvey, Edward P, Richard L. Schaffer and Stephen M. McGarry. "High Performance Fixed-Wing Aircraft Simulation Using SIMNET Protocols." Bolt Beranek and Newman Systems and Technology Library. 1991.

17. Harvey, Edward P. and Richard L. Schaffer. "The Capability of the Distributed Interactive Networking Standard to Support High Fidelity Aircraft Simulation." *Proceddings of the 13th Interservice/Industry Training Systems Conference.* 127-135. 1991.

18. Hobbs, Capt Bruce. 1992. *A User Interface to a True 3-D Display Device.* MS thesis. AFIT/GCE/ENG/92D-06. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1992.

19. *How They Fight Desert Shield Order of Battle Handbook.* Army Intelligence Agency Document AIA-DS-2-90. September 1990.

20. Humphrey, Watts S. *Managing the Software Process.* Reading MA: Addison-Wesley, 1990.

21. Katz, Amnon. "Notes on Dead Reckoning in the DIS Standard." *Summary Report The Sixth Workshop on Standards for the Interoperability of Defense Simulations Volume II: Position Papers.* Prepared for PM TRADE and DARPA by University of Central Florida Institute for Simulation and Training. Document Number IST-CR-92-2. 115-122. 16 April 1992.

22. ——. "Topics in Dynamics" Class handout distributed in AE 468, AE 568, and AE 668, Flight Dynamics and Control. University of Alabama. December 1991.

23. Lippman, Stanley B. *C++ Primer.* Reading MA: Addison-Wesley, 1991.

24. Locke, John, David R. Pratt and Michael J. Zyda. "Integrating SIMNET with NPSNET Using a Mix of Silicon Graphics and Sun Workstations." *Summary Report The Sixth Workshop on Standards for the Interoperability of Defense Simulations Volume II: Position Papers.* Prepared for PM TRADE and DARPA by University of Central Florida Institute for Simulation and Training. Document Number IST-CR-92-2. 161-172. 16 April 1992.

25. McDonald, L. Bruce, et al. "Standard Protocol Data Units for Entity Information and Interaction in a Distributed Interactive Simulation." *Proceedings of the 13th Interservice/Industry Training Systems Conference.* 119-126. 1991.

26. Meliza, Larry L. and Seng Chong Tan. "Application of the SIMNET Unit Performance Assessment System to After Action Reviews." *Proceddings of the 13th Interservice/Industry Training Systems Conference.* 136-144. 1991.

27. *Military Standard (Final Draft) Protocol Data Units and Entity Information and Entity Interaction in a Distributed Interactive Simulation.* Prepared for PM TRADE and DARPA by University of Central Florida Institute for Simulation and Training. Contract Number N61339-91-C-0091. 30 October 1991.

28. Miller, Duncan C., Arthur R. Pope, and Rolland M. Waters. "Long-Haul Networking of Simulators." *Proceedings of the 10th Interservice/Industry Training Systems Conference.* 1988.

29. ——. "The SIMNET Architecture for Distributed Interactive Simulation." Prepared for 1991 Summer Computer Simulation Conference. IDA Simulation Center Library Files Serial Number 50514. July 1991.

30. "Open Systems Interconnection (OSI) Multipeer/Multicast (MPMC) Consortium." *Summary Report The Sixth Workshop on Standards for the Interopcrability of Defense Simulations Volume I: Minutes and List of Attendees.* Prepared for PM TRADE by University of Central Florida Institute for Simulation and Training. Document number IST-CR-92-2. 97-106. March 1992.

31. Pope, Arthur R. *The SIMNET Network and Protocols.* Bolt Beranek and Newman Inc Report Number 7627 prepared for Defense Advanced Research Projects Agency, June 1991.

32. *Rationale Document Entity Information and Entity Interaction in a Distributed Interactive Simulation.* Prepared for PM TRADE by University of Central Florida Institute for Simulation and Training. Document Number IST-PD-92-1. January 1992.

33. Rogers, Capt Brian K., Clarence W. Stephens and Alan B. Oatman. "12th I/ITSC-1990: SIMNET Fighter Aircraft Application." *Proceddings of the 13th Interservice/Industry Training Systems Conference.* 347-356. 1991.

34. Rumbaugh, James, et al. *Object-Oriented Modeling and Design.* Englewood Cliffs NJ: Prentice-Hall, 1991.

35. Saunders, Randy. "Formal Expression of Dead Reckoning: Mathematical and Representation Recommendations." University of Central Florida Institute for Simulation and Training Library Copy.

36. Schaffer, Richard and Rolland Waters *Dead Reckoning Algorithms and the Simulation of High Performance aircraft.* White Paper ASD 91-015. 8 March 1991.

37. *SIMNET 6.6.1+ Network Protocols for the TRUE and WAR BREAKER Programs.* Prepared by Loral Defense Systems-Akron for Air Force Human Resources Laboratory, Williams AFB AZ. Document number AL0692-009 Revision C. 8 September 1992.

38. *SIMNET Data Logger Version 2.2.* Prepared by Bolt Beranek and Newman Systems and Technologies Corportation. October 1989.

39. *SIMNET Semi-Automated Forces: The Combined Arms Workstation User's Guide.* Prepared by Bolt Beranek and Newman Systems and Technologies Corportation for Defense Advanced Research Projects Agency, 30 April 1991.

40. Simpson, Capt Dennis J. 1991. *An Application of the Object-Oriented Paradigm to a Flight Simulator.* MS thesis. AFIT/GCS/ENG/91D-22. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991. (AD-A243624).

41. Smith, Joshua E. *The tblgr Program Version 1.7 User Guide.* Bolt Beranek and Newman Inc. 1992.

42. Stallings, William *Data and Computer Communications.* New York NY: Macmillan Publishing Company, 1988.

43. *Strawman Distributed Interactive Simulation Architecture Description Document Volume I: Summary Description* Prepared by Loral Systems Company for PM TRADE. Document Number ADST/WDL/TR-92-003010. 31 March 1992.

44. Stroustrup, Bjarne *The C++ Programming Language.* Reading MA: Addison-Wesley, 1991.

45. Switzer, John C. 1992. *A Synthetic Environment Flight Simulator: The AFIT Virtual Cockpit.* MS thesis. AFIT/GCE/ENG/92D-17. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1992.

46. Thorpe, Lt Col Jack A. "The New Technology of Large Scale Simulator Networking: Implications for Mastering the Art of Warfighting." *Proceedings of the Ninth Interservice/Industry Training Systems Conference* 492-501. 1987.

47. Zyda, Michael J. et al. "NPSNET: Constructing A 3D Virtual World." *Proceedings 1992 Symposium on Interactive 3D Graphics.* 147-156. 1992.

Vita

Captain Steven Michael Sheasby was born on January 18, 1964 on McClellan AFB, California. He graduated from Holy Cross High School in New Orleans, Louisiana in 1982. In 1986, he graduated magna cum laude from Tulane University in New Orleans, Louisiana with a bachelor of science in computer engineering degree. He was commissioned in May of 1986. His first assignment in the Air Force was located at the headquarters of the Electronic Security Command at San Antonio, Texas. His jobs included computer engineer, program manager, and configuration manager. In 1990, he was assigned to the 6964 Communications-Computer Systems Squadron assigned to headquarters Electronic Security Command which became the 6900 Communications-Computer Systems Group also assigned to headquarters Electronic Security Command. His primary role was intelligence systems programmer/analyst and chair of the group software process enhancement board. He entered the Air Force Institute of Technology in 1991.

Permanent address: 71 Thornton Drive
Chalmette, Louisiana 70043

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE
December 1992

3. REPORT TYPE AND DATES COVERED
Master's Thesis

4. TITLE AND SUBTITLE
MANAGEMENT OF SIMNET AND DIS ENTITIES IN SYNTHETIC ENVIRONMENTS

5. FUNDING NUMBERS

6. AUTHOR(S)
Steven M. Sheasby, Capt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/92D-16

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DARPA/ASTO
3701 North Fairfax Drive
Arlington, Va 22203

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This thesis describes the techniques used to create an object manager utilized by an application program during a distributed interactive simulation. This work is currently utilized by a number of AFIT synthetic environment applications for use during a SIMNET exercise.

An extensive review of distributed interactive simulations is presented. A discussion of the current distributed simulation protocol, SIMNET, is presented along with the future protocol standard, DIS. Finally, a brief discussion on dead reckoning and its importance during an exercise is presented.

An analysis of the SIMNET and DIS protocols provided the basis for the creation of a series of C++ classes to store information on a simulation entity during an exercise. These C++ classes used class generalization and inheritance to differentiate between the different types of entities seen during an exercise.

An entity object manager was developed to perform a set of basic functions required during an exercise as listed in a collection of SIMNET and DIS documents. The entity object manager uses the C++ entity class structure to manage the numerous entities viewed during a typical SIMNET exercise. The entity object manager also communicates with the the other exercise participants using two different government supplied network communications packages.

14. SUBJECT TERMS
Distributed Interactive Simulation, Synthetic Environments, Object-Oriented, Computer Graphics

15. NUMBER OF PAGES
70

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL